SharedSpoons Style Guide

SharedSpoons Team

December 2024

1 Introduction

This document outlines the coding and style conventions for the SharedSpoons React Native project. It serves as a reference to maintain consistency, readability, and scalability in our codebase. All team members are expected to adhere to these guidelines to ensure high-quality software development.

2 Frontend

2.1 File and Folder Organization

To ensure a clear and structured project architecture, our file and folder hierarchy is as follows:

- **tabs/**: Contains individual tab components.
- **pages/**: Contains full-page views, representing screens within the app.
- **components/**: Reusable UI components shared across multiple screens or tabs.
- **constants/**: Defines app-wide constants, such as colors, dimensions, and API endpoints.
- **hooks/**: Custom React hooks encapsulating reusable logic.
- **assets/**: Contains static assets, such as images, fonts, and SVGs.
- **contexts/**: Provides context for managing global state, such as authentication.

2.1.1 Folder Naming Conventions

- Use `kebab-case` for folder names (e.g., `tabs`, `constants`).
- Avoid using spaces or underscores.

2.1.2 File Naming Conventions

- Use `PascalCase` for component and page files (e.g., `HomePage.tsx`).
- Use `camelCase` for utility files and non-component-specific helpers (e.g., `formatDate.ts`).
- Use lowercase and `kebab-case` for assets (e.g., `logo-icon.png`).

2.2 Coding Style and Conventions

We follow the style conventions enforced by ESLint and React Native best practices.

2.2.1 General Guidelines

- Use TypeScript for all files.
- Avoid default exports; prefer named exports for clarity.
- Use `camelCase` for variables, functions, and object properties.
- Use `SCREAMING_SNAKE_CASE` for constants.
- Use `PascalCase` for React components and custom hooks.

2.2.2 Styling Components

- Use `StyleSheet.create()` for React Native styles.
- Organize styles in the same file as the component.
- Use consistent naming for style objects, such as `container`, `button`, and `text`.

2.2.3 Example Component

Below is an example of a typical component adhering to these conventions:

```
import React from 'react';
import { StyleSheet, Text, TouchableOpacity, View } from 'react-native';

interface ButtonProps {
    title: string;
    onPress: () => void;
}

export const Button: React.FC<ButtonProps> = ({ title, onPress }) => {
    return (
        <TouchableOpacity style={styles.button} onPress={onPress}>
            <Text style={styles.buttonText}>{title}</Text>
        </TouchableOpacity>
    );
};

const styles = StyleSheet.create({
    button: {
        backgroundColor: '#007AFF',
        padding: 10,
        borderRadius: 5,
    },
    buttonText: {
        color: '#FFFFFF',
        textAlign: 'center',
    },
});
```

Listing 1: Reusable Button Component

2.3 Linting and Formatting

We strictly enforce linting and formatting rules via:

- **ESLint**: Ensures adherence to JavaScript/TypeScript best practices.

2.3.1 Linting Rules

- Use semicolons at the end of every statement.
- Prefer `const` and `let` over `var`.
- Use arrow functions (`()=>`) for anonymous functions.

3 Backend

3.1 File and Folder Organization

To maintain a clear and structured backend architecture, the file and folder hierarchy is as follows:

- **src/**: Contains the source code for the backend.
  - Each top-level endpoint path (e.g., `auth`, `post`) is a folder within `src/`.
  - Within each endpoint folder:
    - `Controller.ts`: Handles incoming requests and responses.

        * `Service.ts`: Contains logic and interacts with models or external services.
        * `Index.ts`: Contains related type definitions for endpoints.

- **tests/**: Contains unit tests for the backend.
    - Tests are organized by the top-level endpoint path they are testing.
    - Each test file is named as `<top-level path>.test.ts` (e.g., `auth.test.ts`).

3.1.1 Folder Structure

- Organize folders in a way that reflects the API endpoint hierarchy.

3.1.2 File Naming Conventions

- Controller, service, and index files within each endpoint folder must be named exactly:
    - `Controller.ts`
    - `Service.ts`
    - `Index.ts`
- Test files are named as `<top-level path>.test.ts` and placed in the corresponding folder within `tests/`.

3.2 Coding Style and Conventions

Consistent coding style is crucial for collaboration and code quality.

3.2.1 Tabs and Indentation

- Use spaces instead of tabs for indentation.
- Each indentation level consists of **2 spaces**.

3.2.2 Line Length

- Limit all lines of code to a maximum of **100 characters** in length.

3.2.3 Trailing Whitespace

- Ensure there is **no trailing whitespace** at the end of any line.

3.2.4 Unused Variables

- Do not leave any **unused variables** in the codebase.
- Remove or comment out any variables that are declared but not used.

3.3 Linting and Formatting

We strictly enforce linting and formatting rules via:

- **ESLint**: Ensures adherence to JavaScript/TypeScript best practices.

3.3.1 Linting Rules

- Use semicolons at the end of every statement.
- Prefer `const` and `let` over `var`.
- Use arrow functions (`()=>`) for anonymous functions.